PipeStreams: Pipelines and Streams

Terminology

■ **Sources** (a.k.a. 'readables', symbolized as **A**) do not take inputs from the stream, but provide data to 'flow downstream'. Sources may be constructed from values (e.g. a text may be converted into a source of text lines or a series of characters), from files (reading files and processing their contents being a frequent application of streams), or using functions that sit around until they are called with pieces of data they then send downstream.

■ **Sinks** (a.k.a. 'writables', symbolized as **Z**) do not provide outputs to the stream, but accept all data that is funnelled to them by way of the pipeline. There are two important subclasses of sinks: on the one hand, sinks may be used to write to a file, an outgoing HTTP connection, or a database; so that would be sinks with targets. Target-less sinks may seem pointless but they are needed to provide a writable endpoint to a pipeline. The function used to represent such a sink is known as `$drain` in PipeStreams, so 'a drain' just means that, a generic target-less sink required by the API.

■ **Pipelines** (symbolized as **P**) are lists of **stream transforms**. As we will shortly see, pipelines represent the central constructional element in the PipeStreams way of doing things. They start out as generic lists (i.e. Javascript `Array`s) that transforms—functions—are being inserted to, and they end up representing, in the ordering of their elements, the ordering of transformational steps that each piece of data fed into their top end has to undergo before coming out at the other end. A well-written pipeline combines a simple conceptual model with a highly readable list of things to do, with each station along the assembly line doing just one specific thing.

■ **Transforms** (symbolized as $*f*) are synchronous or asynchronous functions that accept data items d_i (a.k.a. events, not to be confused with DOM events or NodeJS `EventEmitter` events) from upstream and pass zero or more data items d_1, d_2, ... down the stream, consecutively.

In short one can say that in each stream, data comes out of a source **A**, flows through a number of transforms $*f*$, and goes into some kind of sink **Z**. A **complete pipeline** has at least a source **A** and a sink **Z**.

■ By **streams** (symbolized as Σ) we mean the activity that occurs when a complete pipeline has been 'activated', that is made start to process data. So, technically, a 'stream' (a composite algorithm at work) is what a 'pipeline', once activated, does. In practice, the distinction is often blurred, and one can, for example, just as well say that a particular event is 'coming down the stream' or 'coming down the pipeline'.

XXX

■ `pull [ source $f(), …, ] -> source`

A pipeline with a source and any number of transforms is equivalent to a source.

■ `pull [ $f(), …, sink ] -> sink`

A pipeline with any number of transforms and a sink is equivalent to a sink.

■ `pull [ $f(), …, ] -> $f()`

A pipeline with any number of transforms is equivalent to a (more complex) transform; in particular,

■ `pull [] -> $pass()`

a pipeline with no elements is equivalent to the empty (no-op) transform that passes all data through.

■ `pull [ source, $f(), …, sink, ] -> Σ`

A pipeline with a source, any number of transforms and a sink is equivalent to a stream.

Simple Example

```
PS  = require 'pipestreams'
log = console.log
p   = []
p.push PS.new_value_source [ 'foo', 'bar', 'baz', ]
p.push PS.$show()
p.push PS.$drain -> log 'done'
```